# New Zealand Informatics Competition 2020
# Round 3 Solutions

January 19, 2021

# Overview

## Questions

1. **Mental A**
2. **Nail Polish**
3. **Bytecoin**
4. **Diversity**
5. **Sort Recovery**

The solutions to these questions are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

## Resources

Ever wondered what the error messages mean?

www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

See our list of other useful resources here:

www.nzoi.org.nz/nzic/resources.html

# Tips for next time

Remember, this is a contest. The only thing we care about is that your code runs. It doesn't need to be pretty or have comments. There is also no need to worry about invalid input. Input will always be as described in the problem statement. For example, the code below is not necessary.

```
# Not needed
def error_handling(prompt):
    while True:
        try:
            N = int(input(prompt))
            if N < 0 or N > 100:
                print('That was not a valid integer!')
            else:
                return N
        except ValueError:
            print('Not a valid integer')
...
```

There are a few other things students can do to improve their performance in contests.

## Practice getting input

A number of students tripped up on processing input with multiple integers on a single line. A neat trick for processing this sort of input in Python is to use the `str.split()` method and the `map()` function. The `split()` method will break up a string at space characters, returning a list of the words. The `map()` function can be used to apply `int()` to each string in this list, converting them to integers. For example, suppose we have the following line of input:

```
1 4 2 7
```

We can turn this into a list of integers with the Python statement

```
my_ints = list(map(int, input().split()))
```

Notice that we used `list()`. This is because `map()` returns us a special generator object, not a list. However, generator objects are easily converted to lists.

We suggest having a go at some of the NZIC Practice Problems.

## Move on to the next question

If you are spending too much time on a question, move on. There could be easy subtasks waiting in the next question. Often, you will think of a solution to your problem while working on another question. It also helps to come back to a question with a fresh pair of eyes.

## Take time to read the questions

Don't underestimate the value of taking time to read and understand the question. You will waste exponentially more time powering off thinking you have the solution only to discover you missed something obvious.

In the real world, it is very rare to be provided with a problem in a simplistic form. Part of the challenge with these contests is reading and understanding the question, then figuring out what algorithm will be needed. Before submitting, check to make sure you have done everything the question asks you to do.

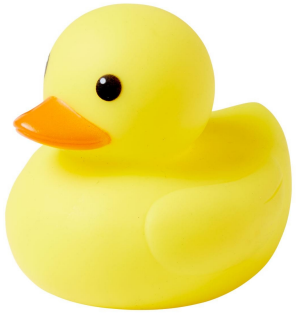
## Test your code first!

It is much more time efficient to test your solutions on your own computer first. It can take a while to wait for the server to run your code. It is far more efficient to test it yourself first, submit it, then start reading the next question while you wait for your previous solution to be marked by the server.

While testing, remember to add some edge case tests. For example, if a question specifies "$1 \leq N \leq 10$" then try out an input where $N = 1$. Think of other tricky inputs that might break your code. Look at the feedback the server is giving you.

## Use a rubber duck

https://en.wikipedia.org/wiki/Rubber_duck_debugging



# Mental A

https://train.nzoi.org.nz/problems/1177

We can keep a variable that stores our answer so far, and then add, subtract, or multiply each input number to that variable. We also need to keep track of which operation we need to apply between the answer so far and the next number. A simple way to do this is with an integer variable, where the value 0 represents addition, 1 represents subtraction, and 2 represents multiplication. We can change the value of this variable after each operation to reflect the next operation that should be performed.

One thing to watch out for is that the first addition is done between the first and second value, and not between the first value and our answer variable. Adding the first value to the answer variable (and then *multiplying* the answer with the second value) was a common bug among some submissions.

## Python Example 1

```
N = int(input())
result = int(input())
state = 0
for i in range(N-1):
    x = int(input())
    if state == 0:
        result = result + x
        state = 1
    elif state == 1:
        result = result - x
        state = 2
    elif state == 2:
        result = result * x
        state = 0
print(result)
```

## Python Example 2

```python
N = int(input())
r = int(input())
for i in range(N-1):
    x = int(input())
    r = [r+x, r-x, r*x][i % 3] # Look into modular arithmetic in Python
print(r)
```

## C++ Example 1

```cpp
#include <iostream>
using namespace std;

int main ()
{
    int N, total=0;
    cin >> N >> total;
    for (int i=0; i < N-1; i++) {
        int num, type = i % 3;
        cin >> num;
        if (type == 0) {
            total += num;
        } else if (type == 1) {
            total -= num;
        } else {  // type == 2
            total *= num;
        }
    }
    cout << total << endl;
}
```

## C++ Example 2

```cpp
#include <iostream>
using namespace std;

int main ()
{
    int N, total=0;
    cin >> N >> total;
    for (int i=0, num=0, type=0; i < N; cin >> num, type = i++ % 3) {
        if (type == 2) total *= num;
```

```
10          else total += type ? -num : num;
11      }
12      cout << total << endl;
13  }
```

# Nail Polish

https://train.nzoi.org.nz/problems/1176

We first check if the colour name of the polish used on the big toe has a repeated letter. A `set` will help us here - if we add all letters for the big toe colour into the `set`, it will only contain the unique letters in that colour. That is, a `set` will get rid of all repeated letters. Therefore, if the number of characters in the `set` is equal to the number of characters in the colour, then there cannot be any repeated letters in the colour as the `set` did not get rid of any. Similarly, if the number of characters in the `set` is less than the number of characters in the colour, then the colour must have repeated characters.

If the big toe colour has have repeated characters, then we check if each of the other four toes has an odd number of letters. If that is the case, then the combination is 'best', otherwise, it must be 'bad'. If the big toe colour does not repeated characters, then we check if the four toe colours are in decreasing length order - that is, that for each toe, the next toe's colour is shorter. If this is the case, then the combination is 'ok', otherwise it is 'bad'.

*Note - to check if a number is odd, we can use the* modulo operator (%)*, which gives the remainder of the division between two numbers. We simply check if our number* modulo *2 is 1, or 0. This operator can also help simplify our logic in the previous problem - can you see how?*

## Python Example

```
a, b, c, d, e = [input() for _ in range(5)]

if len(set(a)) < len(a):
    # first color has repeated character
    if all(len(x) % 2 == 1 for x in [b,c,d,e]):
        print("best")
    else:
```

```
        print("bad")
else:
    # check if lengths in ascending order
    if len(b) > len(c) > len(d) > len(e):
        print("ok")
    else:
        print("bad")
```

## C++ Example

```
#include <iostream>
#include <string>
#include <unordered_set>

using namespace std;

int main ()
{
    string big_toe, other_toe;
    cin >> big_toe;
    unordered_set<char> used_chars;
    for (auto& c : big_toe) {
        // does the character 'c' already exist in the set?
        if (used_chars.count(c)) {
            // repeat found, check for oddness
            for (int i = 0; i < 4; i++) {
                cin >> other_toe;
                // check for even lengths on other toes
                if (other_toe.length() % 2 == 0) {
                    cout << "bad" << endl;
                    return 0;
                }
            }
            cout << "best" << endl;
            return 0;
        }
        used_chars.insert(c);
    }

    // no repeated letters, length must strictly decrease
    int previous_length = 21; // max length = 20 chars
    for (int j = 0; j < 4; j++) {
        cin >> other_toe;
```

```
34          int len = other_toe.length();
35          if (len >= previous_length) {
36              cout << "bad" << endl;
37              return 0;
38          }
39          previous_length = len;
40      }
41      cout << "ok" << endl;
42      return 0;
43  }
```

# Bytecoin

https://train.nzoi.org.nz/problems/1174

Note that since $s_i \leq b_i$ for all $i$, we can never profit from buying and selling a bytecoin on the same day. Thus, we should never sell any bytecoins on the first day, or buy any bytecoins on the last day.

## Subtask 1

For subtask 1, there are always only two days. Clearly, we can make a profit only if $s_2 > b_1$. For each bytecoin we buy, we would make $s_2 - b_1$ dollars of profit. Thus, if $s_2 > b_1$, we should always buy as many bytecoins as we can, and then sell them all on day 2. Otherwise, we cannot make any profit, and our final answer is $C$.

```
N, C = map(int, input().split())
buys = list(map(int, input().split()))
sells = list(map(int, input().split()))
money = max(C, C % buys[0] + C // buys[0] * sells[1])
print(money)
```

## Subtask 2

For subtask 2, the buy prices are equal to sell prices on all days. Let's denote $p_i = s_i = b_i$. Now, consider any two consecutive days, day $x$ and $x + 1$.

If $p_x > p_{x+1}$, then it is never optimal to hold any bytecoins at the end of day $x$. For example, suppose we held $K$ bytecoins at the end of day $x$. Instead, it would be better to sell them for all $p_x$ dollars each and then buy them back the next day at a cheaper price of $p_{x+1}$ dollars each, resulting in a profit of $K \times (p_{x+1} - p_x)$ dollars, while still ending with the same number of bytecoins.

On the other hand, if $p_x < p_{x+1}$, then it is always optimal to hold as many bytecoins as we can at the end of day $x$.

So, for every two consecutive days, if $p_x > p_{x+1}$, we should do nothing. But if $p_x < p_{x+1}$, we should buy as many bytecoins as we can on day $x$, and then sell them all on day $x+1$, for a profit of $\lfloor D/p_x \rfloor \times (p_{x+1} - p_x)$ (where $D$ is the most dollars we can have at the start of day $p_x$). Note that we might end up buying the same number of bytecoins back on day $x+1$, if $p_{x+1} < p_{x+2}$, but since $b_i = s_i$, we do not lose any profit by doing so.

```
N, money = map(int, input().split())
buys = list(map(int, input().split()))
sells = list(map(int, input().split()))
for buy, sell in zip(buys, sells[1:]):
    if buy < sell:
        money += money // buy * (sell - buy)
print(money)
```

## Subtask 3

There are a few different ways to solve this problem, but the simplest approaches use the following observations:

1. **On any given day, we should either buy as many bytecoins as we can, or sell all of our bytecoins, or do nothing.** If we can make profit from buying and selling a single bytecoin, we would clearly be losing profit by not buying the maximal amount of bytecoins.
2. **If it's optimal to buy bytecoins on any given day, we should not buy any more bytecoins until we sell them.** Otherwise, it would have been better to just buy all of our bytecoins on whichever day was cheaper. Similarly, **if it's optimal to sell bytecoins on any given day, we should sell all of them.** If it's optimal to sell a bytecoin, then either (1) we will never buy bytecoins after that, or (2) the next time we buy bytecoins, the buy price must have decreased below the current sell price (otherwise it would have been better to just hold our bytecoin instead of selling it). If the future buy price decreases below the current sell price, then it's obviously better to have sold all of our bytecoins, as we could then buy them back at a cheaper rate.

## An alternative approach

Let's consider the first possible day on which we can sell bytecoins to make profit. Suppose this is day $x$. Then, this must be the first day where the sell price, $s_x$, is greater than the minimum buy price across all previous days, $\min(b_1, b_2, \ldots, b_{x-1})$. Is it optimal to make this trade?

1. Suppose on the next day, that the buy price, $b_{x+1}$, is lower than the current sell price, $s_x$. Then, it must have been optimal to sell on day $x$, as we could then buy back any sold bytecoins for a cheaper price.
2. Suppose on the next day, that the sell price, $s_{x+1}$ increases above the current sell price. Then we would always make more profit selling our bytecoins on day $x + 1$ instead of day $x$.
3. In all other cases, the next buy price must be higher than the current sell price, and the sell price must be lower than the current sell price. We can ignore such days entirely – it is never optimal to buy on day $x + 1$ (as it would have been better to hold on day $x$ instead of selling), and it is never optimal to sell on day $x + 1$ (as we would have made more selling on day $x$ instead).

Our solution keeps track of the minimum buy price we have seen so far (since our last sell), until we reach a higher sell price that would allow us to make profit. We always make profitable trades as soon as we can. To account for the case where the sell price increases before the buy price decreases, we set the initial "minimum buy price" to our sell price, giving us the option to "buy back" our sold bytecoins at no loss.

```
N, money = map(int, input().split())
buys = list(map(int, input().split()))
sells = list(map(int, input().split()))
min_buy = buys[0]
for buy, sell in zip(buys, sells):
    min_buy = min(min_buy, buy)
    if min_buy < sell:
        money += money // min_buy * (sell - min_buy)
        min_buy = sell
print(money)
```

# Diversity

https://train.nzoi.org.nz/problems/1173

## Subtask 1

We can take a brute-force approach, repeatedly incrementing $N$ until it no longer contains any consecutive repeated digits. To do this, we can check for diversity by comparing each character (digit) in the input string to its neighbour. If a digit matches its neighbour, the stirng is not diverse. Otherwise, if no digit matches its neighbour, the number must be diverse. Below is an example.

### Subtask 1 Python Example

```
def is_diverse(N):
    # Create (a, b) pairs for each character paired with its neighbour
    # (offset by 1).
    for a,b in zip(str(N), str(N)[1:]):
        # If a is the same as its neighbour, the string is
        # not diverse so return False
        if a == b:
            return False
    # Otherwise, the string must be diverse
    return True

N = int(input())
while not is_diverse(N):
    N += 1
print(N)
```

It can be proved that for any integer $N$, the smallest diverse number not less than $N$ is always less than $2N$, and since $N$ has $\log_{10} N$ digits, the time complexity of this solution is $O(N \log N)$. For larger $N$, there are much more efficient solutions.

## Subtask 2

In this subtask, all digits of $N$ are guaranteed to be the same. To gain more points, you might use the Subtask 1 solution for $N \leq 100,000$ and handle this special case for larger numbers. Instead of incrementing the number $N$ as for Subtask 1, it more effective to look at the single digits in each place value. For example, let $N_1 = 333333$. Since the first two 3s are the highest place value, we must consider changing one of them. To ensure the next diverse number is as close to $N_1$ as possible, we must leave the first 3 the same. If we increment the first 3 to 4 then we will miss a lower diverse number. Instead, we increment the second 3 so that the next closest diverse number must begin with 34.... Since any choice of the remaining digits will not make the diverse number less than $N_1$, we must choose the lowest value for the remaining digits. In the case of $N_1$, the next lowest diverse number is, therefore, 340101. Generalising this approach leads to a solution for Subtask 2.

However, there is an edge case to consider. For example, $N_2 = 999$. Here, we cannot increment any digits in-place. So, if $N$ is a number of all 9s, the next lowest diverse number needs an extra digit of the next highest place value. This operation is often called a 'carry'. The next highest diverse number of $N_2$ would, therefore, be 1010. Using the Subtask 1 solution for $N <= 100,000$ and this approach for Subtask 2, you can obtain a score of 50% on this problem.

## Subtask 3

We can find a diverse number that is greater than or equal to $N$ by only looking at the individual digits of $N$. As for Subtask 2, we must consider the case where resolving a repeat number 9 requires us to carry an increment to the next highest place value. However, unlike Subtask 2, the digits are not all the same. Hence, we might arrive at a case where $N_3 = 9899$. Here, the two highest place value digits 98... are different. Instead, it is the two lowest place value 9s which requires us to increment the 8 to a 9 which then creates a repeated value. The next diverse number is, therefore, 10101 for this case.

There are a number of ways that students solved this problem. For example, a program can look for a repeating digit, fix it while increasing the number's value as little as possible, then 'restart' the search for repeating digits again. Since, in the worst case, we might change a digit every two digits or so, the number of restarts is proportional to the number of digits, $\log_{10} N$. From the last digit we change, the remaining digits are set to the lowest value combination of diverse digits, which is a repeating pattern of 0 and 1. The overall time complexity for this approach is

$O\left(\log^2 N\right)$, which is still acceptable if we are only talking in the order of 10 digits. Some coded examples are given below.

Another solution is to start at the lowest place value digit (the end digit) and work backwards towards the highest place value digit. If we encounter a repeat digit, we attempt to increment the lowest place value of the two repeat digits we find. However, if that digit is a 9, we must carry the increment to the next highest place value. If the next highest place value is also a 9, we must continue the carry to the next highest place value. If incrementing the next highest place value via a carry also creates a repeated digit (as with $N_3$) then we must attempt to increment again, and carry if we cannot increment. You may realise, though, that stopping here would give a number that is too high and also potentially leave behind repeat digits which are unresolved, meaning the result is not a diverse number. Indeed, it is only the changed digit of highest place value (the last one we were forced to increment) that we care about. The remaining digits of lower place value are replaced with an alternating 0101.. pattern such to minimise the value of the diverse number. If we change no digits during our scan then $N$ is already diverse, so we need not change any digits. This approach is $O\left(\log N\right)$ time, which is slightly better, but perhaps more difficult to write. For this problem the improvement does not make a difference but for super long numbers (such as in Diversity 2) it does. See the latter examples below for how this logic might be implemented.

## Subtask 3 Python Example 1

```
N = input()
while True:
    for idx in range(len(N)-1):
        # find duplicates
        if N[idx] == N[idx+1]:
            # fix duplicate
            # Use the int() function to handle overflows and
            # set the remaining digits to all 0
            pair_index = idx + 2
            N = str(int(N[:pair_index])+1) + "0"*(len(N)-pair_index)

            # Restart the search for duplicates
            break
    else:
        # If the string has no duplicates, we are finished
        break
print(N)
```

## Subtask 3 C++ Example 1

```cpp
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string num;
    cin >> num;

    auto it = num.begin();
    while (it < num.end()-1) {
        // find duplicate
        if (*it == *(it+1)) {
            int origonal_length = num.size();
            int pair_index = it-num.begin() + 2;

            // fix duplicate
            num = to_string(std::stoi(num.substr(0, pair_index)) + 1);
            int carry = num.size() - pair_index;

            // pad the rest with zeros, to be fixed later
            num.resize(origonal_length + carry, '0');

            // restart
            it = num.begin();
        } else {
            // keep searching
            it++;
        }
    }
    cout << num << endl;
}
```

## Subtask 3 Python Example 2

```python
N = list("0" + input()). # Add leading zero for overflow
turn = len(N) # Remember location of last changed digit
carry = False # Carry an increment to a higher place value

# Iterate backwards
for i in range(len(N)-1, -1, -1):
    if N[i] < '9':
        same = i > 0 and N[i-1] == N[i]
        # For repeat characters (and carries) then increment the digit
        if carry or same:
            N[i] = chr(ord(N[i]) + 1)
            turn = i+1
            # Continue to carry?
            if carry and i > 0:
                carry = N[i-1] == N[i]
    elif N[i-1] == '9':
        carry = True

# Set all digits below the last changed digits to '010..' pattern
for j in range(turn, len(N)):
    N[j] = chr(ord("0") + (N[j-1] == '0'))

# Don't print leading zero if it remains
print("".join(N[1:]) if N[0] == '0' else "".join(N))
```

## Subtask 3 C++ Example 2

```cpp
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string num;
    cin >> num;

    num = "0" + num;
    auto turn = num.end();
    bool carry = false;

```

```
15      for (auto it = num.rbegin(); it < num.rend(); it++) {
16          // Can we increment?
17          if (*it < '9') {
18              // Should we increment?
19              if (carry || (it != num.rend()-1 && *(it+1) == *it)) {
20                  *it += 1;
21                  // Convert from reverse iterator to normal
22                  // iterator with .base(). Usually a +1 is
23                  // needed, but because the 0/1 conversion
24                  // below works from the next lowest place
25                  // value we also -1 which cancles to 0.
26                  turn = it.base();
27                  // If incrementing creates a duplicate then
28                  // we must carry the increment to the next
29                  // place value.
30                  if (carry && it != num.rend()-1) carry = *(it+1) == *it;
31              }
32          } else if (*(it+1) == '9') {
33              // A 9 cannot be incremented so carry the
34              // increment to the next higher place value.
35              carry = true;
36          }
37      }
38
39      // Work back for all place values lower than the highest
40      // place value we changed and replace with the minimal
41      // alternating 0/1 pattern. E.g. 33999999 -> 34010101.
42      for (;turn < num.end(); turn++) {
43          *turn = '0' + (*(turn-1) == '0');
44      }
45
46      // Don't print leading zero if it remains
47      if (num[0] == '0') {
48          cout << num.substr(1, num.size()-1) << endl;
49      } else {
50          cout << num << endl;
51      }
52  }
```

# Sort Recovery

https://train.nzoi.org.nz/problems/1178

## Subtask 1

We first need to identify the primary key: the first column used to sort the rows. We know that this primary column must be in sorted order, otherwise another column must have been the primary key. We can check whether a column is sorted in a particular order by checking if every adjacent pair or values in the column is in that order, or is 'tied' (has two equal values). If we apply this to both columns, we can determine which is sorted, and therefore which is the primary key (if both columns are sorted, then we can pick either of them to be the primary key).
Now, we only need to determine whether the secondary column was used to sort the rows in ascending or descending order. Note that the secondary column's sort order would only have been applied to pairs of rows that were 'tied' in the primary column - rows that are already sorted in the primary column will not be affected by the secondary column's sort order. To find the secondary column's sort order, we go through every group of such tied rows and check their values in the secondary column; if those not tied, then their sort order must be the sort order of the secondary column as a whole. If all pairs in the secondary column are tied, then it does not matter whether it was ascending or descending - it would not have affected the table, so we can choose either order.
For an example, consider the table:

| row | col1 | col2 |
|---|---|---|
| 0 | 2 | 5 |
| 1 | 2 | 5 |
| 2 | 2 | 4 |
| 3 | 1 | 3 |
| 4 | 1 | 3 |

The column 1 must be the primary key, as it is in descending order (each value is less than or equal to the previous value, so each adjacent pair is in descending order). Then, there are three pairs of tied rows: rows (0 & 1), (1 & 2), and (3 & 4). The first pair gives us no information on the ordering of column 2, as the corresponding values in column 2 are also tied at a value of 5. However, the second tied pair, rows 1 & 2, has a descending pair in column 2: 5, followed by 4. Thus, its sort order must be descending, as the descending order broke the tie between rows 1 and 3. Since a valid sort order is guaranteed, we don't need to check any remaining pairs, as we know they must follow the same pattern.

## Python Subtask 1 Solution

```
R, C = map(int, input().split())
table = []
for row in range(R):
    table.append(list(map(int, input().split())))

# Find primary column and its order
if all(table[r+1][0] >= table[r][0] for r in range(R-1)):
    primary_col = 0
    print(1, "asc")
elif all(table[r+1][0] <= table[r][0] for r in range(R-1)):
    primary_col = 0
    print(1, "desc")
elif all(table[r+1][1] >= table[r][1] for r in range(R-1)):
    primary_col = 1
    print(2, "asc")
else:
    primary_col = 1
    print(2, "desc")

secondary_col = 1 if primary_col == 0 else 0

for r in range(R-1):
    if table[r+1][primary_col] == table[r][primary_col]:  #If pair of
    ↪ rows is tied in the primary column
        if table[r+1][secondary_col] < table[r][secondary_col]: # Check
        ↪ if pair is descending in secondary column
            print(secondary_col+1, "desc")
            break
else: # Look into for/else loop in Python
    # No pairs were descending, so must be ascending
```

```
29      print(secondary_col+1, "asc")
```

## Subtask 2

In this subtask, $R$ and $C$ are both very small. Thus, we can simply try out every possible sort order and see if it is valid. To check if a sort order is valid, we need to attempt to sort the table according to that sort order. If the resulting order of the rows matches their order in the original table, then we know that this sort order is correct; otherwise, it is invalid (note that we are guaranteed that at least one of the options we try will be valid).
This leads to a $\mathcal{O}(C! * 2^C * (C * R \log R))$ algorithm, as we try every possible order of columns ($C!$ possible permutations), and for each of those, every possible choice of ascending / descending orderings ($2^C$ possible combinations). The $C * R \log R$ factor is from sorting the table each time ($R \log R$ for sorting complexity, multiplied by $C$ for the number of elements in each row that need comparing). With the worst case of $R == C == 5$, this gives about 100,000 operations, which is well within a one second time limit.

### Python Subtask 2 Solution

```
1  import itertools
2  import copy
3
4  R, C = map(int, input().split())
5  table = []
6  for row in range(R):
7      table.append(list(map(int, input().split())))
8  original_table = copy.deepcopy(table)
9
10 A = itertools.permutations(range(C))
11 B = itertools.product([1, -1], repeat=C)
12 for order, directions in itertools.product(A, B):
13     for i,j in zip(order, directions):
14         table.sort(key=lambda x: x[i]*j)
15     if original_table == table:
16         for i,j in zip(reversed(order), reversed(directions)):
17             print(i+1, ["asc", "desc"][j == -1])
18         break
```

# Subtask 3

From Subtask 1, we know how to check if a column can be used as the primary key, and that if multiple choices are available, we can pick any one of them.
We can continue similarly to find every following key in order. For the secondary key, we only need to check if the column is sorted within each pair of rows that have the same value in the primary key column. In other words, for adjacent rows that have different primary key values, the order of their secondary key values is irrelevant, as it is only used in case of ties. Only the pairs of rows are still tied so far require the secondary (or subsequent) keys to break the tie, so we only need to check those pairs of rows.
This leads to an $\mathcal{O}(R*C^2)$ algorithm, where we repeatedly scan all the columns to find one that can be picked as the next sort key. Finding the next sort key requires us to check every remaining column, and check whether it would be a valid choice. To check validity, we must go through every as-yet tied pair of rows in the column and check whether it is sorted in our current column. Once we find a column that 'breaks' a remaining tie, that column must be the next sort key. We continue until we have found the ordering of every column.
This means that finding the next sort key is $\mathcal{O}(C*R)$, and we must repeat this $C$ times (to get all $C$ columns) for a $\mathcal{O}(C^2*R)$ solution.

## C++ Subtask 3 Solution

```cpp
#include <bits/stdc++.h>
using namespace std;

int R,C;
vector<vector<int>> table;

int main() {
    scanf("%d%d",&R,&C);
    table.resize(R, vector<int>(C));
    for(int i = 0; i < R; i++) {
        for(int j = 0; j < C; j++) {
            scanf("%d",&table[i][j]);
        }
    }
    set<int> unused_cols; // The columns for which we haven't determined a sort order/priority
    for(int i = 0; i < C; i++) unused_cols.insert(i);

```

```
    // Currently tied rows. An integer r in this vector represents the adjacent pair of rows (r, r+1)
    vector<int> tied_rows;
    for(int i = 0; i < R-1; i++) tied_rows.push_back(i);

    while(!unused_cols.empty()) {
        int col = -1;
        for(int c : unused_cols) {
            // If every tied row is in ascending order
            if(all_of(tied_rows.begin(), tied_rows.end(), [&c](int r){ return table[r+1][c] >= table[r][c]; })) {
                // The row must be the next key, and sorted ascending
                cout << c+1 << " asc" << endl;
                col = c;
                break;
            }
            // If every tied row is in descending order
            if(all_of(tied_rows.begin(), tied_rows.end(), [&c](int r){ return table[r+1][c] <= table[r][c]; })) {
                // The row must be the next key, and sorted descending
                cout << c+1 << " desc" << endl;
                col = c;
                break;
            }
        }
        // We have now determined the order of this column, remove it from unused_cols
        unused_cols.erase(col);
        // The pairs of rows that were tie-broken by this column are not longer tied
        // so should not be considered in the next iteration.
        // next_rows will contain only the pairs of rows that are still tied
        vector<int> next_rows;
        for(int r : tied_rows) {
            if(table[r+1][col] == table[r][col]) next_rows.push_back(r);
        }
        tied_rows = next_rows;
    }
}
```

## Subtask 4

To efficiently detect when a column becomes valid as a sort key, we can keep a count of the number of adjacent values in the column that are in ascending order, and the number of adjacent values that are in descending order. When either counts reach 0, we can use that column as a sort key - all adjacent values are now either all ascending or all descending, so the entire column is sorted. Note that these counts only involve the adjacent values that still need tie-breaking, as explained in subtask 3.

Every time we find the next sort key column, we iterate over the pairs of adjacent rows that have different values in that column. Because we only care about the pairs that still need tie-breaking, we can now ignore those un-tied pairs, and for each remaining column, we subtract 1 from its appropriate counter if the corresponding pair of values in the column was out of order. We need to track which adjacent pairs of rows have been processed this way and only process each pair once, to avoid double-subtracting. One way we can do this is to use a list of pairs of rows that are still valid to be processed in the future. For each column, the list consists of all currently tied pairs, and we then process only the pairs in the list on the next column.

It will take $C$ iterations of the main loop to order all $C$ columns. In each iteration, we will loop through at most $C$ other columns to find the next valid key. Additionally, in each iteration, we will loop through every still-tied pair of rows (of which there are at most $R$). So far, this gives us a complexity of $\mathcal{O}(C * (R + C))$. For some of the rows each iteration, we will also have to loop through every column to update the counts. However, for any row, we can do this only once, as this only occurs when the row first becomes un-tied, so will not be processed in later iterations. Thus, we only perform at most $\mathcal{O}(R * C)$ count-updates throughout the entire program. This ensures the time complexity is $\mathcal{O}(C * (R + C) + R * C)$ $= \mathcal{O}(C * (R + C))$.

## C++ Full Solution

```cpp
#include <bits/stdc++.h>
using namespace std;

int R,C;
vector<vector<int>> table;

int main() {
    // Read input
    scanf("%d%d",&R,&C);
```

```
    table.resize(R, vector<int>(C));
    for(int i = 0; i < R; i++) {
        for(int j = 0; j < C; j++) {
            scanf("%d",&table[i][j]);
        }
    }

    set<int> unused_cols; // The columns for which we haven't determined a sort order/priority
    for(int i = 0; i < C; i++) unused_cols.insert(i);

    // Currently tied rows. An integer r in this vector represents the adjacent pair of rows (r, r+1)
    vector<int> tied_rows;
    for(int i = 0; i < R-1; i++) tied_rows.push_back(i)

    vector<int> asc_counts(C);
    vector<int> desc_counts(C);

    // Initialise ascending/descending counts for every row
    for(int r = 0; r < R-1; r++) {
        for(int c = 0; c < C; c++) {
            if(table[r+1][c] > table[r][c]) asc_counts[c]++;
            if(table[r+1][c] < table[r][c]) desc_counts[c]++;
        }
    }

    while(!unused_cols.empty()) {
        int col = -1;
        // Find a column that can be used as the next key
        for(int c : unused_cols) {
            if(desc_counts[c] == 0) {   // No descending pairs, so column must be in ascending order
                cout << c+1 << " asc" << endl;
                col = c;
                break;
            }
            if(asc_counts[c] == 0) {    // No ascending pairs, so column must be in descending order
                cout << c+1 << " desc" << endl;
                col = c;
                break;
            }
```

```
49          }
50          assert(col != -1);
51          unused_cols.erase(col);
52          vector<int> next_rows;
53          for(int r : tied_rows) {
54              // If the pair is still tied, add it to the rows that must
                ↪  be checked next iteration
55              if(table[r+1][col] == table[r][col]) next_rows.push_back(r);
56              else {  // Row is no longer tied
57                  for(int c : unused_cols) {
58                      if(table[r+1][c] > table[r][c]) asc_counts[c]--;
                        ↪  // Pair is in descending order, so decrement
                        ↪  column's ascending count
59                      if(table[r+1][c] < table[r][c]) desc_counts[c]--;
                        ↪  // Pair is in ascending order, so decrement
                        ↪  column's descending count
60                  }
61              }
62          }
63          tied_rows = next_rows;
64      }
65  }
```

# Big O complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list $N$ items long. We often call the measured parameter $N$. For example, a list of length $N$.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million ($100\,000\,000$) operations per second. For example, if the problem specifies a time limit of 1 second and an input of $N$ as large as $100\,000$, then you know that an $O(N^2)$ algorithm might be too slow for large $N$ since $100\,000^2 = 10\,000\,000\,000$, or 10 billion operations.

## Time complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a+b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as $N$ increases (i.e. does not depend on $N$). For example, say you have a program containing a list of $N$ items and want to access the item at the $i$-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry $i$ corresponds to, but these will take the same amount of computation regardless of $N$. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O\,(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a couple of extra operations every time

$N$ doubles in size.[1] For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double $N$ to 16. As far as efficiency goes, this is pretty good, since $N$ generally has to get very, very large before a computer starts to struggle.

$O\left(\log^2 N\right)$ time. This was found to be a suitable algorithm complexity for solving the Diversity problem. It is worse than $O(\log N)$ for large numbers of digits, but the maximum number of digits in Diversity is only 10 so this is fine. For very very large numbers, as in Diversity 2, this does make a difference. If we define $K$ to be $K = \log_{10} N$ (the number of digits) then this complexity is equivalent to $O\left(K^2\right)$, instead of $O\left(K\right)$ as required for Diversity 2.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to $N$. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length $N$ to confirm you have indeed found the maximum. Usually, if you have one loop that iterates $N$ times your algorithm is $O(N)$.

$O\left(N^2\right)$ time means the number of operations is proportional to $N^2$ . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of $N$ items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of $N$, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10\,000$.

[1] More formally, it means there exists some constant $c$ for which the program takes at most $c$ extra operations every time $N$ doubles in size.